Table of Contents

Introduction 1.1
引言 1.2
特性和用途 1.2.1
JSSE 标准 API 1.2.2
SunJSSE 供应商 1.2.3
相关文档 1.2.4
JRE 安装目录 1.3
术语和定义 1.4
SSL 协议总览 1.5
为何使用 SSL 1.5.1
SSL 如何工作 1.5.2
JSSE 类和接口 1.6
核心类和接口 1.6.1
支持类和接口 1.6.2
次要支持类和接口 1.6.3
自定义 JSSE 1.7
TLS 重新协商问题 1.8
硬件加速和智能卡的支持 1.9
Kerberos密码套件 1.10
额外的 Keystore 格式(PKCS12) 1.11
SNI 扩展 1.12
问题解决 1.13
代码示例 1.14
将不安全的 Socket 转为安全的 Socket 1.14.1
运行 JSSE 示例代码 1.14.2
使用 JSSE 创建 Keystore 1.14.3
使用 SNI 扩展 1.14.4

Java Secure Socket Extension (JSSE) Reference Guide 《JSSE 参考指南》

This is a Chinese translation of Java Secure Socket Extension (JSSE) Reference Guide, and also provides a lot of useful examples about JSSE.

本书是《JSSE 参考指南》的中文翻译，同时提供了大量 JSSE 实例，帮助你快速理解 JSSE 的用法和原理。

本书利用业余时间编写，由于时间紧凑，精力和能力有限，书中未免有纰漏和错误，望读者能够热忱斧正，点此提问。如有兴趣，也可以参与到本翻译工作中来 :)

另外有 GitBook 的版本方便阅读 http://waylau.gitbooks.io/jsse-reference-guide。

书中所有实例，在https://github.com/waylau/jsse-reference-guide `samples` 目录下。

从目录开始阅读吧！

Contact 联系作者:

- Blog:www.waylau.com
- Gmail: waylau521@gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

引言

数据在网络上传播很容易被不是预期的收件人进行访问。当数据包含私人信息，如密码和信用卡号码，必须采取措施，以保证使数据不容易透露给未经授权的第三方。同样重要的是，在传输过程中要确保数据没有被修改（有意或无意地）。 Secure Sockets Layer (SSL，安全套接字层)和Transport Layer Security (TLS，传输层安全)协议，旨在保护数据在网络传输的过程中的保密性和完整性。

Java Secure Socket Extension (JSSE，Java 安全套接字扩展) 实现了安全的互联网通信。它提供了一个框架，并为执行 SSL 和 TLS 协议的Java 版本的实现，包括数据加密，服务器认证，消息完整性和可选的客户端认证功能。使用 JSSE，开发者可以提供一个客户端和运行任何基于 TCP/IP 上的应用协议（如 HTTP，Telnet 或 FTP）的服务器之间的数据安全通道。关于 SSL 的更多介绍，请参阅 SSL 协议总览。

通过抽象复杂的底层安全算法和握手机制，JSSE 减少创造微妙而危险的安全漏洞的风险。此外，作为一个模块，开发人员可以直接集成它到他们的应用程序简化了应用程序的开发。

JSSE 提供了一个应用程序接口（API）的框架和该 API 的实现。该 JSSE API 补充了核心网络，并通过 java.security 和 java.net 包中提供的扩展的网络套接字类，如信任管理，密钥管理器，SSL 上下文和用于定义的加密服务、套接字工厂框架来封装套接字创建行为。因为 SSLSocket 类是基于阻塞 I/O 模型，Java 开发工具包（JDK）包括一个非阻塞的SSLEngine 类，来选择实现适合自己的 I/O 方法。

该 JSSE API 能够支持 SSL 2.0 和 3.0 版和 TLS 1.0 版。这些安全协议封装一个正常的双向流套接字和 JSSE API 增加了认证，加密和诚信保障的支持。 JSSE 跟 JDK 实现了支持 SSL 3.0 和 TLS 1.0。它不实现SSL 2.0。

JSSE 是 Java SE 平台的安全组件，并且在其他基于 Java Cryptography Architecture (JCA，基于 Java 加密体系结构）框架找到相同的设计原则。该框架加密技术有关的安全组件可以让他们实现独立，并尽可能的实现算法独立。 JSSE 使用由 JCA 框架中定义的加密服务供应商。

在 Java SE 平台的其他安全组件包括 Java Authentication Authorization Service (JAAS，Java认证和授权服务) 和 Java Security Tools （Java安全工具）。 JSSE 包含许多相同的概念和算法的 JCA 但自动适用于下面一个简单的流套接字 API。

该 JSSE API 的目的是让其他的 SSL/TLS 协议，公钥基础设施（PKI）实现无缝的插入。开发者也可以提供替代的逻辑来确定是否远程主机应信任什么或验证密钥材料是否应该被发送到远程主机。

特性和用途

JSSE 包括以下重要的特性:

- 作为一个标准的 JDK 组件
- 可扩展的,基于供应商的架构
- 100% 纯 Java 实现
- 提供了支持SSL版本 2.0 和 3.0,TLS 1.0 和以后版本的API，以及SSL 3.0 、TLS 3.0的实现和
- 包含可以实例化创建安全通道(SSLSocket、SSLServerSocket SSLEngine)的类
- 支持作为 SSL 握手的一部分的密码套件协商,用来启动或验证安全通信
- 支持客户端和服务器身份验证,这是正常的 SSL 握手的一部分
- 支持 HTTP 封装 SSL 协议,它允许使用 HTTPS 访问如 web 页面数据
- 提供服务器会话管理 api 来管理驻留内存的 SSL 会话
- 支持几种加密算法中常用密码套件,其中包括下表中列出的：

Table 1: Cryptographic Functionality Available in JSSE

密码算法[注1]	加密过程	密钥长度(字节)		
Rivest Shamir Adleman (RSA)	认证和密钥交换	512 以及更大Rivest Cipher 4 (RC4)	批量加密	128，128 (40 effective)
Data Encryption Standard (DES)	批量加密	64 (56 effective)，64 (40 effective)		
Triple DES (3DES)	批量加密	192 (112 effective)		
Advanced Encryption Standard (AES)	批量加密	256[注2] ，128		
Diffie-Hellman (DH)	密钥协议	1024，512		
Digital Signature Algorithm (DSA)	身份验证	1024		

- 注1. SunJSSE 的实现是使用 JCA 用于所有的加密算法 ↩
- 注2. 密码套件使用 AES_256 需要安装 Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files。 详见 Java SE下载页面。 ↩

JSSE 标准 API

JSSE API标准,可在 javax.net 和 javax.net.ssl 包找到,提供如下:

- 安全套接字和服务器套接字
- 非阻塞引擎用于生产和消费 SSL/TLS 数据流(SSLEngine)
- 工厂创建套接字、服务器套接字、SSL套接字,和SSL服务器套接字。通过使用套接字工厂,可以封装套接字创建和配置行为
- 一个类代表一个安全套接字上下文作为工厂用于安全套接字工厂和引擎
- 密钥和信任管理器接口(包括特定的 X.509 密钥和信任管理器),和工厂,可以用于创建它们
- 一个类用于安全 HTTP URL 连接(HTTPS)

SunJSSE 供应商

Oracle 的 Java SE 的 JSSE 实现是包含了一个名为 SunJSSE 的供应商，通过 JCA 来预装和预注册。这个供应商提供以下加密服务:

- SSL 3.0 和 TLS 1.0 安全协议的实现
- 最常见 SSL 和 TLS 密码套件的实现,其中包括认证、密钥协议、加密和完整性保护
- 一个基于 x.509 的密钥管理器的实现,用于从标准的 JCA keystore 中选择合适的认证密钥
- 一个基于 x.509 的基于信任管理器的实现，用于证书链路径验证规则
- PKCS12 实现了 JCA keystore 类型 "pkcs12"。PKCS12 不支持存储信任锚。用户应该存储信任锚在 Java keystore(JKS) 格式并保存私钥为 PKCS12 格式。

更多关于这个提供者的信息可见 SunJSSE 文档

相关文档

这个是本主题相关的文档和书名：

- JSSE API 文档
  - javax.net
  - javax.net.ssl
  - javax.security.cert
- Java SE Security
  - Java SE Security 文档主页
  - Java SE Security 主页
  - Java 教程中的 The Security Features in Java SE
  - Java PKI 编程者指南
  - Java 2 Platform Security, Second Edition: Architecture, API Design and Implementation
- Cryptography
  - Dr. Ronald L. Rivest 的 Cryptography and Security 页面 (不再维护)
  - Applied Cryptography, Second Edition by Bruce Schneier. John Wiley and Sons, Inc., 1996.
  - Cryptography Theory and Practice by Doug Stinson. CRC Press, Inc., 1995. Third edition published in 2005.
  - Cryptography & Network Security: Principles & Practice by William Stallings. Prentice Hall, 1998. Fifth edition published in 2010.
- Secure Sockets Layer (SSL)
  - The SSL Protocol version 3.0 Internet Draft
  - The TLS Protocol Version 1.0 RFC
  - HTTP Over TLS RFC
  - SSL and TLS: Designing and Building Secure Systems by Eric Rescorla. Addison Wesley Professional, 2000.
  - SSL and TLS Essentials: Securing the Web by Stephen Thomas. John Wiley and Sons, Inc., 2000.
  - Java 2 Network Security, Second Edition, by Marco Pistoia, Duane F Reller, Deepak Gupta, Milind Nagnur, and Ashok K Ramani. Prentice Hall, 1999.
- U.S. Encryption Policies
  - U.S. Department of Commerce: http://www.commerce.gov/
  - Computer Systems Public Policy http://www.techceocouncil.org/
  - Federal Information Processing Standards Publications (FIPS PUBS): http://csrc.nist.gov/publications/PubsFIPS.html
  - Revised U.S. Encryption Export Control Regulations:

http://epic.org/crypto/export_controls/regs_1_00.html

JRE 安装目录

java-home 变量占位符的使用是为了引用 Java Runtime Environment (JRE，Java运行时环境) 的安装目录。这个目录的确认是基于有或没有安装 JDK 的 JSSE 的运行来判断的。JDK 包括 JRE,但位于不同的文件层次结构中。

java-home 的默认位置如下表：

操作系统	JDK	JRE
Solaris/Linux	~/jdk1.8.0/jre	~/jre1.8.0
Windows	C:\jdk1.8.0\jre	C:\jre1.8.0

注：在路径名的波浪线 (~) 是代表在 Solaris，Linux 或 Mac OS X操作系统的当前用户的主目录。

术语和定义

本文档包含如下术语和定义：

authentication (身份验证)

The process of confirming the identity of a party with whom one is communicating.

确认与人交流的过程中一方的身份。

cipher suite(密码套件)

A combination of cryptographic parameters that define the security algorithms and key sizes used for authentication, key agreement, encryption, and integrity protection.

通过加密参数的组合来定义的安全算法和密钥大小，用于身份验证、密钥协商、加密和完整性保护。

certificate（证书）

A digitally signed statement vouching for the identity and public key of an entity (person, company, and so on). Certificates can either be self-signed or issued by a Certificate Authority (CA) â€” an entity that is trusted to issue valid certificates for other entities. Well-known CAs include VeriSign, Entrust, and GTE CyberTrust. X509 is a common certificate format that can be managed by the JDK's keytool.

数字签名的声明用来标识一个实体的身份和公钥(个人,公司,等等)。证书可以是自签名或是一个证书颁发机构(CA) 来颁发。证书是证明其他实体可以信任的有效证件。众所周知的 CA 包括 VeriSign,Entrust, 和 GTE CyberTrust。X509 证书是一种常见的格式,可以由 JDK 的 keytool 管理。

cryptographic hash function（加密散列函数）

An algorithm that is used to produce a relatively small fixed-size string of bits (called a hash) from an arbitrary block of data. A cryptographic hash function is similar to a checksum and has three primary characteristics: it is a one-way function, meaning that it is not possible to produce the original data from the hash; a small change in the original data produces a large change in the resulting hash; and it does not require a cryptographic key.

这是一个算法,用于从任意的数据块产生一个相对较小的部分固定大小的字符串(称为哈希)。加密散列函数类似于一个校验和,有三个主要特征:它是一种单向函数,这意味着它是不可能产生的原始数据散列,一个小的改变原始数据产生很大的变化产生的散列;它不需要密钥。

Cryptographic Service Provider（加密服务供应商）

Sometimes referred to simply as provider for short, the Java Cryptography Architecture (JCA) defines it as a package (or set of packages) that implements one or more engine classes for specific cryptographic algorithms. An engine class defines a cryptographic service in an abstract fashion without a concrete implementation.

有时简称为供应商,Java Cryptography Architecture (JCA,Java加密体系结构)将它定义为一个包(或一组包)实现一个或多个引擎类特定的加密算法。一个引擎类定义了一个抽象的方式而没有具体实现的加密服务。

decryption (解密)

See encryption/decryption.

见下面的加密/解密。

digital signature（数字签名）

A digital equivalent of a handwritten signature. It is used to ensure that data transmitted over a network was sent by whoever claims to have sent it and that the data has not been modified in transit. For example, an RSA-based digital signature is calculated by first computing a cryptographic hash of the data and then encrypting the hash with the sender's private key.

数字相当于一个手写签名。它用于确保数据在网络上传输声明是由谁发送它,并且确保数据在传输过程中没有被修改。例如,一个基于 RSA 数字签名计算,首先计算密码散列的数据,然后用发送方的私钥加密散列。

encryption/decryption（加密/解密）

Encryption is the process of using a complex algorithm to convert an original message (cleartext) to an encoded message (ciphertext) that is unintelligible unless it is decrypted. Decryption is the inverse process of producing cleartext from ciphertext.

The algorithms used to encrypt and decrypt data typically come in two categories: secret key (symmetric) cryptography and public key (asymmetric) cryptography.

加密的过程中使用复杂的算法来将一个原始消息(明文)转为编码信息(密文),除非它是解密不然编码信息是不可识别的。解密是从密文生成明文的逆过程。

算法用于加密和解密数据通常有两类:密钥(对称)加密和公钥(不对称)加密。

handshake protocol（握手协议）

The negotiation phase during which the two socket peers agree to use a new or existing session. The handshake protocol is a series of messages exchanged over the record protocol. At the end of the handshake, new connection-specific encryption and integrity protection keys are generated based on the key agreement secrets in the session.

协商阶段期间,两个套接字对同意使用新的或现有的会话。握手协议是一系列交换记录协议的消息。握手结束的时候,新的特定的连接加密和完整性保护密钥的生成是基于会话中的密钥协商密钥。

key agreement（密钥协商）

A method by which two parties cooperate to establish a common key. Each side generates some data, which is exchanged. These two pieces of data are then combined to generate a key. Only those holding the proper private initialization data can obtain the final key. Diffie-Hellman (DH) is the most common example of a key agreement algorithm.

一个双方合作的方法来建立一个共同的密钥。每一方生成一些数据,并进行交换。这些两个数据会结合生成一个密钥。只有那些持有适当的私人初始化数据可以获得最后的密钥。Diffie-Hellman (DH) 是密钥协议算法最常见的例子。

key exchange（密钥交换）

A method by which keys are exchanged. One side generates a private key and encrypts it using the peer's public key (typically RSA). The data is transmitted to the peer, who decrypts the key using the corresponding private key.

密钥交换的一种方法。一方生成私钥,并使用对等的公钥加密它(通常是RSA)。数据传输到对等端,使用相应的私钥解密这个密钥。

key manager/trust manager（密钥管理器/信任管理器）

Key managers and trust managers use keystores for their key material. A key manager manages a keystore and supplies public keys to others as needed (for example, for use in authenticating the user to others). A trust manager decides who to trust based on information in the truststore it manages.

密钥管理器和信任管理器使用 keystore 作为密钥的原料。密钥管理器管理 keystore 并根据需要向他人提供公钥(例如,用于验证用户)。信任管理器基于 truststore 的信息来决定谁是可以信任的。

keystore/truststore

keystore is a database of key material. Key material is used for a variety of purposes, including authentication and data integrity. Various types of keystores are available, including PKCS12 and Oracle's JKS.

keystore 是一个数据库的密钥材料。密钥材料是用于各种各样的用途,包括身份验证和数据的完整性。有各种类型的 keystores 可用的,包括PKCS12 和 Oracle JKS。

Generally speaking, keystore information can be grouped into two categories: key entries and trusted certificate entries. A key entry consists of an entity's identity and its private key, and can be used for a variety of cryptographic purposes. In contrast, a trusted certificate entry contains only a public key in addition to the entity's identity. Thus, a trusted certificate entry cannot be used where a private key is required, such as in a javax.net.ssl.KeyManager. In the JDK implementation of JKS, a keystore may contain both key entries and trusted certificate entries.

一般来说,keystore 信息可以分为两类:密钥条目和可信任证书条目。一个密钥条目包含一个实体的身份和其私钥,而且可以用于各种加密的目的。相比之下,一个受信任的证书条目除了实体的身份只包含公钥。因此,不能使用可信证书条目,私钥是必需的,比如 javax.net.ssl.KeyManager。JDK 实现 JKS,keystore 可能只包含密钥条目和可信任证书条目。

A truststore is a keystore that is used when making decisions about what to trust. If you receive data from an entity that you already trust, and if you can verify that the entity is the one that it claims to be, then you can assume that the data really came from that entity.

一个 truststore 也是一个 keystore, 作用是决定什么是值得信任。如果你从一个实体已经接收数据,你已经信任了,如果你能确认该实体就是它声称是那个,那么你可以假定的数据来自那个实体。

An entry should only be added to a truststore if the user trusts that entity. By either generating a key pair or by importing a certificate, the user gives trust to that entry. Any entry in the truststore is considered a trusted entry.

一个条目只能是该用户信任的实体才能被添加 truststore。通过生成一个密钥对或通过导入证书,用户给予该条目以信任。信任存储库中的任何条目被认为是一个值得信赖的条目。

It may be useful to have two different keystore files: one containing just your key entries, and the other containing your trusted certificate entries, including CA certificates. The former contains private information, whereas the latter does not. Using two files instead of a single keystore file provides a cleaner separation of the logical distinction between your own certificates (and corresponding private keys) and others' certificates. To provide more protection for your private keys, store them in a keystore with restricted access, and provide the trusted certificates in a more publicly accessible keystore if needed.

有两个不同的 keystore 文件:一个包含只是你的密钥条目,另外一个包含你信任的证书条目,包括 CA 证书。前者包含私人信息,而后者没有。使用两个文件代替单个 keystore 文件提供了一个更整洁的分离的逻辑区别自己的证书(和相应的私钥)和其他人的证书。为了提供更多保护私匙,将它们存储在有访问限制的 keystore 里面,并在需要的时候可以提供更公开访问 keystore 的可信证书。

message authentication code (MAC) 消息身份验证代码

Provides a way to check the integrity of information transmitted over or stored in an unreliable medium, based on a secret key. Typically, MACs are used between two parties that share a secret key in order to validate information transmitted between these parties.

提供了一种基于密钥方法来检查信息传输或存储在一个不可靠的介质的完整性。通常,使用两方之间的 MAC 来共享一个密钥来验证这些双方之间的信息传输。

A MAC mechanism that is based on cryptographic hash functions is referred to as HMAC. HMAC can be used with any cryptographic hash function, such as Message Digest 5 (MD5) and Secure Hash Algorithm (SHA), in combination with a secret shared key. HMAC is specified in RFC 2104.

MAC 机制是基于加密散列函数称为 HMAC。HMAC 可用于任何的加密散列函数，如 Message Digest 5 (MD5，消息摘要5) 和 Secure Hash Algorithm (SHA，安全散列算法),结合一个秘密共享密钥。HMAC 在 RFC 2104 中指定。

public-key cryptography 公开密钥加密

A cryptographic system that uses an encryption algorithm in which two keys are produced. One key is made public, whereas the other is kept private. The public key and the private key are cryptographic inverses; what one key encrypts only the other key can decrypt. Public-key cryptography is also called asymmetric cryptography.

加密系统使用加密算法的生成两个密钥。一个密钥是公开,而另一个是私有的。公钥和私钥是加密逆;一个密钥加密只有另一个密钥才能解密。公开密钥加密也被称为非对称加密。

Record Protocol 记录协议

A protocol that packages all data (whether application-level or as part of the handshake process) into discrete records of data much like a TCP stream socket converts an application byte stream into network packets. The individual records are then protected by the current encryption and integrity protection keys.

协议包的所有数据(应用程序级还是握手过程的一部分)转变成离散的数据记录，就像一个 TCP 流套接字转化成应用程序字节流到网络数据包。独立的记录被当前的加密和完整性保护密钥保护 。

secret-key cryptography 密钥加密

A cryptographic system that uses an encryption algorithm in which the same key is used both to encrypt and decrypt the data. Secret-key cryptography is also called symmetric cryptography.

使用的加密算法的加密系统都使用相同的密钥来加密和解密数据。密钥加密技术也称为对称加密。

session 会话

A named collection of state information including authenticated peer identity, cipher suite, and key agreement secrets that are negotiated through a secure socket handshake and that can be shared among multiple secure socket instances.

命名的状态信息集合包括验证对等身份,密码套件和密钥协商，主要通过安全套接字协议的秘密握手来协商,可以在多个安全套接字实例之间共享。

trust manager 信任管理器

See key manager/trust manager.

见 密钥管理器/信任管理器

truststore

See keystore/truststore.

见 keystore/truststore

SSL 协议总览

Secure Sockets Layer (SSL，安全套接字层)是在网络上应用最广泛的加密协议实现。SSL 使用结合加密过程来提供网络的安全通信。本节介绍 SSL 和它所使用的加密过程。

SSL 提供了一个安全的增强标准 TCP/IP 套接字用于网络通信协议。如表3所示,添加了安全套接字层传输层和应用层之间的标准 TCP/IP 协议栈。SSL的应用程序中最常用的是 Hypertext Transfer Protocol (HTTP，超文本传输协议),这个是互联网网页协议。其他应用程序,如 Net News Transfer Protocol (NNTP，网络新闻传输协议)、Telnet、 Lightweight Directory Access Protocol (LDAP，轻量级目录访问协议), Interactive Message Access Protocol (IMAP，互动信息访问协议)和 File Transfer Protocol (FTP,文件传输协议),也可以使用 SSL。

注:目前还没有标准的安全的 FTP。

Table 3: TCP/IP Protocol Stack with SSL

TCP/IP 层	协议
Application Layer	HTTP, NNTP, Telnet, FTP, 等
Secure Sockets Layer	SSL
Transport Layer	TCP
Internet Layer	IP

SSL 是由网景公司在 1994 年,来自互联网社区的投入,现在已经演变成为一个标准。现在国际标准组织 Internet Engineering Task Force (IETF)的控制下。IETF 更名为 SSL 为 Transport Layer Security (TLS，传输层安全),在 1999年1月发布了第一个规范,版本1.0。TLS 1.0 对于 SSL 的最新版本 3.0版本 是一个小的升级。SSL 3.0 和 TLS 1.0之间的差异是微小的。TLS 1.1 是在2006年4月发布的,TLS 1.2 在 2008年8月 发布。然而,这些更新的版本并不像 TLS 1.0 和 SSL 3.0 一样广泛支持。

为何使用 SSL

通过网络传输敏感信息是有风险的，出于以下原因:

- 你不能总是确保与你交流的实体真的是你认为那位。
- 网络数据可以被截获,因此它是有可能被未经授权的第三方（有时被称为一个攻击者）所读取。
- 攻击者截获数据并可能修改它才将其发送给接收者。

SSL 解决了这些问题。它解决了第一个问题的方式是,有选择地允许通讯双方的身份来确保另一方在一个身份验证过程一旦双方认证通过，SSL 提供了一个双方消息安全传输的加密连接。加密双方之间的通信提供了隐私,因此解决第二个问题。加密算法使用 SSL 包括一个安全散列函数,这是类似于一个校验和。这将确保在传输过程中数据没有修改。安全散列函数解决了第三个问题，即数据完整性。

注意:身份验证和加密都是可选的,依赖于两个实体之间的协商密码套件。

电子商务的交易就是一个明显的使用 SSL 的例子。在电子商务交易,也许会愚蠢的认为你能保证与你交流的对方服务器的身份。企业别人很容易就能创建一个虚假网站并提供各种方法来让你输入你的信用卡号码。SSL 可以允许客户端来验证服务器的身份。它还允许服务器对客户机进行身份验证的身份,虽然在网络交易中,人们很少能做到这一点。

一旦客户机和服务器都熟悉了彼此的身份,SSL 通过加密算法来提供隐私和数据完整性。这使得敏感信息,比如信用卡号码,通过互联网传输会变得安全。

尽管 SSL 提供身份验证、隐私和数据完整性,它不提供”不可抵赖性”服务。
Nonrepudiation（不可抵赖性）意味着一个实体发送一条消息后不能否认发送过它。当数字相当于一个与消息有关的签名,那么发送消息之后还是可以证明发送过。SSL 本身没有提供不可抵赖性。

SSL 如何工作

SSL 是有效的原因之一是,它使用几种不同的加密过程。SSL 使用公开密匙加密方式来提供身份验证和密钥加密技术与数字签名来提供隐私和数据的完整性。在你能理解 SSL 之前,它有助于理解这些加密过程。

加密过程

密码学的主要目的是让一个未经授权的第三方难以访问和理解双方私人之间的通信。虽然并不总是限制所有未经授权的访问数据,但私人数据应可以通过加密的过程的对于未授权方来说应该是隐蔽的。加密使用复杂的算法将原始消息(明文)编码为加密信息(密文)。加密和解密的算法在网络上传输的数据,通常有两类:密钥加密和公开密钥加密。下面将详细介绍这些形式的加密。

密钥加密和公开密钥加密依赖达成一致的密钥的使用或密钥对。一个密钥是一个字符串使用的密码算法或算法过程中加密和解密的数据。密钥就像锁的钥匙,只有正确的钥匙可以打开锁。

交际双方之间安全地传输的密钥并不是一件小事。公钥证书使一方能够安全地传输它的公钥,同时确保接收者的公钥的真实性。公钥证书在后面一节中描述。

加密过程的描述,使用安全社区广泛使用的约定:交际双方都贴上名字 Alice 和 Bob 。未经授权的第三方,即攻击者,叫 Charlie。

密钥加密技术

在密钥加密技术中,交流的双发 Alice 和 Bob,使用相同的密钥来加密和解密消息。在加密的数据可以通过网络发送之前,Alice 和 Bob 都必须拥有密码，并且他们将用于加密和解密的密码算法必须是一致的。

密钥加密技术的一个主要问题是如何保证密钥是从一方传递到另一个，且中间确保不允许攻击者访问。如果是用密钥加密技术保护他们的数据,如果 Charlie 获得访问他们的密钥,那么 Charlie 就能理解从 Alice 和 Bob 之间拦截的任何秘密信息。Charlie 不仅可以解密 Alice 和 Bob 的消息,而且他也可以假装他是 Alice 来向 Bob 发送加密数据。Bob 将不知道该消息来自 Charlie,不是 Alice。

一旦解决了密钥分配的问题,密钥加密技术还是一个有价值的工具。算法提供优秀的安全性并且加密数据的速度相对迅速。 大多数时候在一个 SSL 会话发送使用密钥加密技术发送敏感数据。

密钥加密技术也称为对称加密,因为使用相同的密钥来加密和解密数据。众所周知的秘密密钥加密算法包括 Data Encryption Standard (DES), Triple DES (3DES), Rivest Cipher 2 (RC2), 和 Rivest Cipher 4 (RC4)

公开密钥加密

公开密钥加密的使用一个公钥和一个私钥来解决密钥分发问题。公钥可以通过公开网络发送,而私钥是由通信一方的保管。公钥和私钥的作用是相反的，一个用于加密,另一个用于解密。

假设 Bob 想发送一个秘密消息 Alice ，则他使用公钥加密。Alice 有一个公钥和一个私钥,所以她把她的私钥在一个安全的地方,然后把公钥传给 Bob。Bob 用 Alice 的公钥来加密给 Alice 消息。Alice 稍后就能用她的私钥来解密该消息。

如果 Alice 使用自己的私钥加密消息并发送加密消息给 Bob,Bob 可以确保他收到的数据来自 Alice；如果 Bob 可以用 Alice 的公钥解密数据,则说明该消息一定是被 Alice 用她的私钥加密,并且只有 Alice 有 Alice 的私钥。问题是,任何人都可以读取这个消息,因为 Alice 的公钥是公开的。尽管这个场景不允许安全的数据通信,但它提供了数字签名的基础。数字签名是公钥证书的一个组成部分,用于 SSL 对客户机或服务器进行身份验证。在后面部分会介绍了公钥证书和数字签名。

公开密钥加密也被称为 asymmetric cryptography（非对称加密）,因为不同的密钥用于加密和解密数据。常用与 SSL 的一个著名的使用公钥加密算法是 Rivest Shamir Adleman (RSA) 算法。另一个 Diffie-Hellman (DH) 算法常用于 SSL 的密钥交换。公开密钥加密需要大量的计算,所以会非常缓慢。因此通常仅用于加密小块的数据,如密钥,而不是批量加密数据通信。

比较密钥加密和公开密钥加密

密钥加密和公开密钥加密各有优点和缺点。密钥加密技术,可以快速加密和解密数据,但由于通信双方都必须共享相同的密钥信息,密钥的交换可能是一个问题。公开密钥加密,密钥交换并不是一个问题,因为不需要公钥保密的,但是算法用于加密和解密数据需要大量的计算,因此非常缓慢。

公钥证书

公钥证书提供了一个安全的方法，在非对称加密中传递公开密钥。公钥证书，避免了以下情况：如果 Charlie 创造了他自己的公钥和私钥，他可以伪装他是 Alice 并发送公钥给 Bob。Bob 将能够与 Charlie 通讯，但Bob 会认为他是把他的数据传给了 Alice。

一个公钥证书可以被认为是一个数字相当于一个护照。它是由一个值得信任的组织 Certificate Authority (CA，证书颁发机构)发出，并提供身份的识别。CA 的一个值得信赖的组织。可以比作一个公证人。为了从 CA 中获得一个证书，必须提供身份证明。一旦 CA 确认了申请人就是代表它所述的组表，则 CA 就能签署包含了有效的信息的证书。

公钥证书包含以下字段：

- Issuer （发行者）： CA 是发行证书的。如果用户信任 CA 发行的证书，并且证书是有效的，那么用户可以信任证书
- Period of validity（有效期）： 证书是有失效日期的。当验证证书的有效性时，需检查这个日期。
- Subject： 包括证书代表的实体信息。
- Subject's public key （Subject 的公钥）： 证书提供的主要信息是 Subject 的公钥。提供所有其他字段,是为了确保这个密钥的有效性。
- Signature（签名）： 证书由 CA 进行数字签名后颁发。证书签名是使用 CA 私钥创建的，并确保证书的有效性。因为只有证书是签名的，而在 SSL 中传输的数据并没有，所以 SSL 不提供 nonrepudiation （不可抵赖性）。

因为 Bob 只接受 Alice 发送在公钥证书的有效的公钥，因此当 Charlie 伪装成 Alice时， Bob 不会被欺骗而发送秘密信息给 Charlie 。

证书链中可链接多个证书。当使用证书链时，第一个证书始终是发送者的。下一个是实体证书颁发的发布者的证书。如果有更多的证书在链中，那么每个是颁发前一个证书的权限。链中的最后一个证书是一个 CA 的证书。根 CA 是一个被广泛信任的公共证书颁发机构。一般的，根 CA 的信息是存储在客户端的互联网浏览器。此信息包括 CA 的公钥。众所周知的 CA 包括VeriSign, Entrust, 和 GTE CyberTrust。

加密散列函数

发送加密数据时,SSL 通常使用加密散列函数来确保数据的完整性。散列函数阻止 Charlie 篡改 Alice 发送给 Bob 的数据。

密码散列函数类似于一个校验和。主要的区别在于,一个校验和的目的是检测意外改变数据,而加散列函数是用来检测刻意改变。当数据是由加密散列函数处理,将会生成一小串字符串,称为一个散列。一个微小的改变通常会使生成的散列发生大的改变。加码散列函数不需要密钥。SSL 经常使用的两个散列函数是 Message Digest 5 (MD5) 和 Secure Hash Algorithm (SHA)。SHA 是由 U.S. National Institute of Standards and Technology (NIST) 提出。

消息验证码

消息验证码（MAC）是类似于加密散列，除了它是基于一个密钥。当密钥信息被包含密码散列函数处理的数据，然后得到的散列被称为一个 HMAC。

如果 Alice 希望确保 Charlie 没有篡改 她发送给 Bob 的消息，那么她就可以计算一个她消息的 HMAC，并追加 HMAC 到她的原始消息中。然后，她可以使用共享给 Bob 密钥加密附加了 HMAC 的消息。当 Bob 解密该消息，并计算 HMAC，他就能被告知消息在传输过程中是否被修改。在使用SSL 时，HMAC 是用于加密数据的传输。

数字签名

一旦加密散列创建一个消息,这个散列是用发送者的私钥加密的。这个加密的散列称为数字签名。

SSL 握手

使用 SSL 通信始于一个客户端和服务器之间的信息交换。这种信息交流被称为 SSL 握手。SSL 握手包括以下几个阶段：

1.协商密码套件

SSL 会话开始于一个客户端与服务器之间的协商，他们将使用的密码套件。密码套件是一组加密算法和密钥大小，可以用来加密数据。该密码套件包括公共密钥交换算法或密钥协商算法的信息，以及加密散列函数。客户端告诉服务器哪些是可用的密码套件，服务器选择最可接受的密码套件。

2.认证服务器的身份（可选）

在 SSL，认证步骤是可选的，但在电子商务交易的例子，在网上，客户一般会想验证服务器。认证服务器允许客户端确认服务器代表实体就是客户认为的那个。

为了证明一个服务器属于它声称的组织，服务器向客户端提供其公钥证书。如果这个证书是有效的，那么客户端可以确定服务器的身份。

客户端和服务器交换信息，允许他们在相同的密钥上达成一致。例如，与RSA，客户端使用从公钥证书的取得服务器的公钥，来加密密钥信息。客户端向服务器发送加密的密匙信息。只有服务器才能解密该消息，因为该解密所需的服务器私钥是加密该消息的。

3.同意加密机制

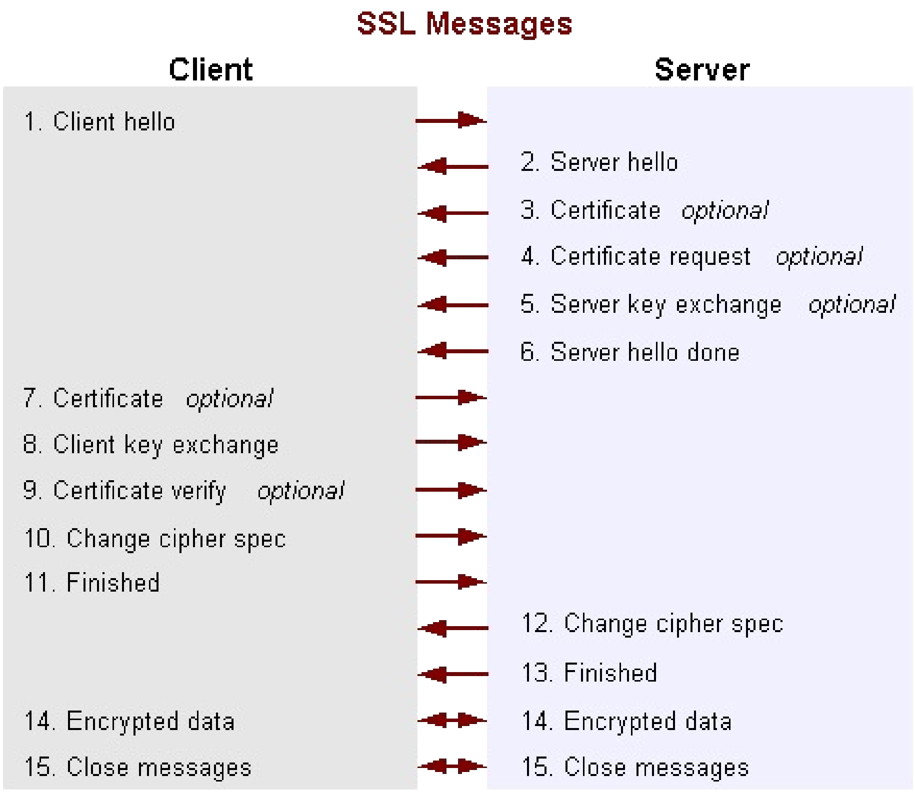
客户端和服务器都可以访问同一个密钥。每一个消息，他们都使用的加密散列函数和在握手的第一步选择和加密散列算法以及共享的秘密信息，来计算附加到消息 HMAC。然后他们使用密钥和在握手的第一步秘协商的密钥算法来对安全数据和 HMAC 加密。客户端和服务器就可以安全的使用他们的加密和散列数据进行交互了。

SSL 协议

前一节中提供的 SSL 握手的一个高层次的描述，这是发送加密信息之前，客户端和服务器之间的信息交换。本节提供更多细节。

图1显示的是在SSL握手交换消息的序列。仅在某些情况下被发送的消息是 optional （可选的）。每个 SSL 信息如下图。

Figure 1: Sequence of Messages Exchanged in SSL Handshake



The SSL messages are sent in the following order:

1.Client hello

客户端发送服务器的信息，包括最高版本的SSL，它支持一个密码套件列表，它支持（如 TLS 1.0 代表 SSL 3.1）。密码套件的信息包括加密算法和密钥大小。

2.Server hello

服务器选择SSL最高版本和最好的密码套件，客户端和服务器都支持并将此信息发送到客户端。

3.Certificate (optional) 证书

服务器给发送客户端证书或证书链。证书链通常从服务器的公钥证书开始，并以证书颁发机构的根证书结束。此消息是可选的，但在需要服务器身份验证时使用。

4.Certificate request (optional) 证书请求

如果服务器必须对客户端进行身份验证，那么它就会向客户端发送证书请求。在互联网应用中，这一消息很少被发送。

5.Server key exchange (optional) 服务器密钥交换

如果证书的公钥信息不足以用于密钥交换，服务器向客户端发送一个服务器密钥交换信息。例如，在基于 Diffie-Hellman (DH)密码套件，该消息包含服务器的 DH 公钥。

6.Server hello done

服务器告诉客户端已经完成了初步协商消息。

7.Certificate (optional) 证书

如果服务器请求来自客户端的证书，那么客户端发送它的证书链给服务器，正如服务器之前做的一样。

注：只有少数几个互联网服务器应用程序要求客户端的证书。

8.Client key exchange 客户密钥交换

该客户端生成用于创建一个密钥以用于对称加密的信息。对于 RSA 来说，然后客户机用服务器的公钥加密密钥信息并将其发送给服务器。密码套件基于DH，该消息包含客户机的 DH 公钥。

9.Certificate verify (optional) 证书验证

该消息由客户端发送，客户端提供如前所述证书。其目的是允许服务器以完成认证所述客户端。当使用此消息时，客户端发送的信息将使用加密散列函数进行数字签名。当服务器用客户端的公钥解密该信息时，该服务器就能够验证客户端。

10.Change cipher spec 更改加密规范

客户端发送消息告诉服务器更改为加密模式。

11.Finished 完成

客户端告诉服务器它准备好可以进行安全的数据通信。

12.Change cipher spec 更改加密规范

服务器发送消息告诉客户端更改为加密模式。

13.Finished 完成

服务器告诉客户端它准备好可以进行安全的数据通信了。这个就是 SSL 握手的截至。

14.Encrypted data 加密数据

客户端和服务器使用对称加密算法和在 client hello 与 server hello 过程中使用加密散列函数协商，并使用客户端在客户端密钥交换过程中发送到服务器的密钥。握手可以在这个时候重新协商。详情见下节。

15.Close Messages 关闭消息

在连接的最后，每个侧发送一个 close_notify 消息来通知对等端连接被关闭了。

如果 SSL 会话期间生成的参数被保存，然后这些参数有时可以再用于将来的SSL 会话。保存 SSL 会话参数允许加密通信开始的更加迅速。

再次握手（重新协商）

一旦初始握手完成，应用程序数据流转，任何一方都可以随时启动新的握手。一个应用程序可能需要使用一个更强大的加密套件，特别是关键操作，或者一个服务器应用程序可能需要验证客户端身份。

不管原因，新的握手是在现有的加密会话中进行的，并且在一个新的会话建立之前，应用数据和握手信息是交错的。

您的应用程序可以使用下列方法之一启动一个新的握手：

- SSLSocket.startHandshake()
- SSLEngine.beginHandshake()

请注意，相关谈判协议的缺陷在 2009 年被发现。协议和 Java SE 的实现都是修正过的。更多信息，查看Transport Layer Security (TLS) Renegotiation Issue

密码套件选择与远程实体验证

SSL/TLS 协议定义了一系列具体的措施以保证受保护的连接。然而，密码套件的选择直接影响到连接所享有的安全类型。例如，如果选择一个匿名密码套件，则该应用程序无法验证远程对等端的身份。如果一个没有加密的套件被选中，那么数据的隐私就不能被保护。此外，SSL/TLS 协议不指定凭据接收到的必须匹配所期望发送的。如果连接被某种方式重定向到一个恶意的对等端，但该流氓的凭据基于目前的信任材料是可以接受的基础上，那么该连接将被认为是有效的。

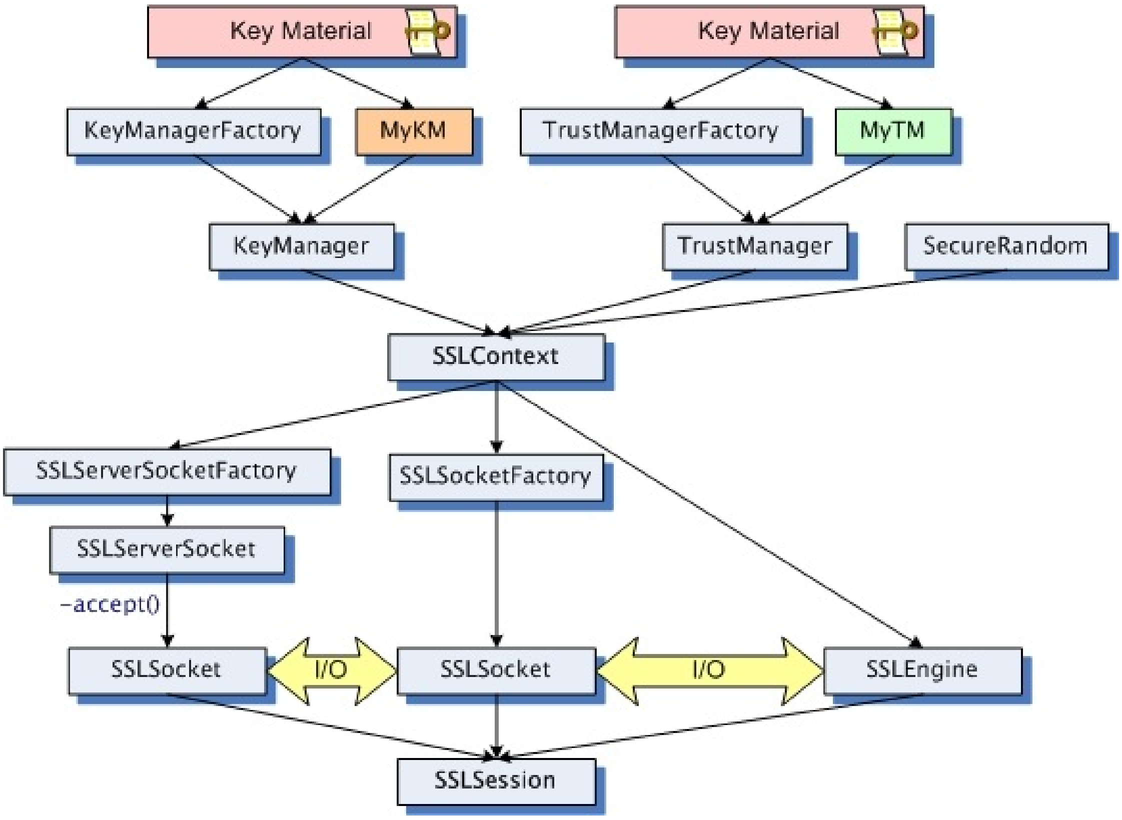
利用原生 SSLSocket 和 SSLEngine 类的时候，你应该总是在发送任何数据前校验凭据。SSLSocket 和 SSLEngine 类不自动验证 URL 中的主机名和凭证中的的主机名是否匹配。如果注解名未被检验，则一个应用程序可以被伪装的 URL 所恶意利用。

协议如 HTTPS(HTTP Over TLS) 需要主机名验证。应用程序可以使用 HostnameVerifier 覆盖默认 HTTPS 主机名称规则。更多信息见 HttpsURLConnection 。

JSSE 类和接口

为了安全地通信，连接双方必须启用 SSL。在 JSSE API，端点类的连接是SSLSocket 和 SSLEngine。在图2中，用于创建 SSLSocket 和 SSLEngine 的主要类是按逻辑顺序摆放。图下面的文字，解释了图中的内容。

Figure 2: Classes Used to Create SSLSocket and SSLEngine



核心类和接口

支持类和接口

次要支持类和接口

自定义 JSSE

TLS 重新协商问题

硬件加速和智能卡的支持

Kerberos密码套件

额外的 Keystore 格式(PKCS12)

SNI 扩展

问题解决

代码示例

下面是本节包含代码示例：

- 将不安全的 Socket 转为安全的 Socket
- 运行 JSSE 示例代码
- 使用 JSSE 创建 Keystore
- 使用 SNI 扩展

将不安全的 Socket 转为安全的 Socket

本节提供的源代码的例子来说明如何使用 JSSE 将不安全的 Socket 连接转为安全的 Socket 连接。本节中的代码摘自本书 Java SE 6 Network Security（Marco Pistoia 等著）。

第一个例子是“没有 SSL 的 Socket 实例”的示例代码，可以使用不安全的 Socket 设置客户端和服务器之间的通信。此代码是在“使用 SSL 的 Socket 实例”上的例子做了修改。

没有 SSL 的 Socket 实例

下面是服务器端和客户端建立不安全 socket 连接。

在一个 Java 程序中，作为服务器和客户端使用 socket 交互，建立 socket 通讯类似是以下的代码：

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;

ServerSocket s;

try {
    s = new ServerSocket(port);
    Socket c = s.accept();

    OutputStream out = c.getOutputStream();
    InputStream in = c.getInputStream();

    // Send messages to the client through
    // the OutputStream
    // Receive messages from the client
    // through the InputStream
} catch (IOException e) { }
```

客户端使用 socket 来与服务器通讯，代码设置如下：

```
import java.io.*;
import java.net.*;

. . .

int port = availablePortNumber;
String host = "hostname";

try {
    s = new Socket(host, port);

    OutputStream out = s.getOutputStream();
    InputStream in = s.getInputStream();

    // Send messages to the server through
    // the OutputStream
    // Receive messages from the server
    // through the InputStream
} catch (IOException e) { }
```

使用 SSL 的 Socket 实例

下面是服务器端和客户端建立安全 socket 连接。

在一个 Java 程序中，作为服务器和客户端使用安全的 socket 交互，建立 socket 通讯类似是以下的代码：

```
import java.io.*;
import javax.net.ssl.*;

. . .

int port = availablePortNumber;

SSLServerSocket s;

try {
    SSLServerSocketFactory sslSrvFact =
        (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    s = (SSLServerSocket)sslSrvFact.createServerSocket(port);

    SSLSocket c = (SSLSocket)s.accept();

    OutputStream out = c.getOutputStream();
    InputStream in = c.getInputStream();

    // Send messages to the client through
    // the OutputStream
    // Receive messages from the client
    // through the InputStream
}

catch (IOException e) {
}
```

户端使用安全的 socket 来与服务器通讯，代码设置如下：

运行 **JSSE** 示例代码

该 JSSE 示例程序来说明如何应用 JSSE 到：

- 创建一个客户端和一个服务器之间的安全套接字连接
- 创建一个 HTTPS 网站的安全连接
- 使用与 RMI 安全通信
- 举例说明 SSLEngine 使用

当您使用示例代码，请注意示例程序的目的是演示如何使用JSSE。它们的目标不是设计成强大的应用程序。

注：建立安全通信涉及到复杂的算法。示例程序提供了没有反馈的设置过程中。当你运行的程序，要有耐心：您可能看不到了任何输出。如果您用设置程序所有 javax.net.debug 系统属性，你会看到更多的反馈。为了介绍阅读本调试信息，请参阅调试 SSL/TLS 连接。

哪里可以找到示例代码

大部分示例代码位于示例文档的同一目录中的示例子目录中。按照该链接以查看所有示例代码文件和文本文件的列表。该页面还提供了一个链接到一个ZIP文件，您可以下载来获取所有的示例代码的文件.

以下部分描述了示例。欲了解更多信息，请参阅README.txt文件。

示例代码说明客户端和服务器之间的安全 **Socket** 连接

在 samples/sockets 目录下的示例程序说明了如何设置客户端和服务器之间的安全 socket 连接。

当运行样本客户端程序，可以与现有的服务器进行通信，例如商业 Web 服务器，也可以与示例服务器程序，ClassFileServer 交互。您可以连接到同一个网络中不同的机器上运行示例客户端和服务器示例程序，也可以在一台机器上，但来自不同的终端窗口中运行它们。

所有 samples/sockets/client 目录下的 SSLSocketClient 示例（以及 *Sample Code Illustrating HTTPS Connections* 中描述的 *URLReader* ）可以与 ClassFileServer 运行。Running SSLSocketClientWithClientAuth with ClassFileServer 描述了如何使用这个例子。对 ClassFileServer 做类似的改动来运行 URLReader，SSLSocketClient，或 SSLSocketClientWithTunneling。

如果客户机和服务器之间通信过程中发生了认证错误（无论是使用网络服务器或 ClassFileServer），最有可能的原因是为必要的密钥没有在 truststore（信任密钥数据库）。例如，ClassFileServer 使用一名为testkeys 的 keystore 包含用于 localhost 进行 SSL 握手的

私钥 。testkeys keystore 包含在相同的 samples/sockets/server 目录作为 ClassFileServer 源。如果客户端无法找到 localhost 在 truststore 中对应的公钥证书，就会产生验证错误。一定要使用samplecacerts truststore （包含 localhost 的公钥和证书），正如在下一节中所述。

配置要求

运行示例程序，创建一个安全的客户机和服务器之间的连接时，您将需要确保相应的证书文件（truststore）可用。对于客户端和服务器程序，你应该使用 samples 目录的证书文件 samplecacerts。使用该证书文件将允许客户端对服务器进行身份验证。该文件包含了所有常用的 Certificate Authority (CA) 使用 JDK（在 cacerts 文件）的证书，加上一个与 ClassFileServer 通讯时由客户端需要验证 localhost 的证书。Classfileserver 使用包含了 localhost 私钥的 keystore，对应于samplecacerts 的公钥。

为了让 samplecacerts 文件提供给客户端和服务器，你可以将它复制到java-home/lib/security/jssecacerts ，重命名为 cacerts，用它来代替 java-home/lib/security/cacerts 文件，或添在 java 命令行形式加以下选项在 客户端和服务器中：

```
-Djavax.net.ssl.trustStore=path_to_samplecacerts_file
```

关于 java-home 的更多信息，参见JRE安装目录。

为了让 samplecacerts truststore 密码修改。你可以使用 keytool 工具在你自己的证书示例中。

如果您使用浏览器，如 Netscape Navigator 或微软的IE浏览器，访问ClassFileServer 实例中的 SSL 服务器，然后一个对话框会弹出的消息，说它不承认证书。这是很正常的，因为与示例程序一起使用的证书是自签名的，并且只用于测试。您可以接受当前会话的证书。经过测试SSL服务器，你应该退出浏览器，从浏览器的命名空间删除测试证书。

对于客户端身份验证，一个单独的 duke 证书是在适当的目录中可用。公钥和证书存储在 samplecacerts 文件。

运行 SSLSocketClient

SSLSocketClient.java 程序演示如何创建一个客户端,使用一个 SSLSocket 发送一个 HTTP 请求和从一个 HTTPS 服务器响应。这个项目的输出是 https://www.verisign.com/index.html 的 HTML 源代码。

你不能在提供防火墙保护的情况下运行这个程序，不然会得到一个UnknownHostException，因为 JSSE 通过防火墙访问www.verisign.com 找不到路径。创建一个等效的客户端可以运行在防火墙保护相爱,需要设置代理隧道见，示例程序SSLSocketClientWithTunneling。

运行 SSLSocketClientWithTunneling

SSLSocketClientWithTunneling.java 程序说明了如何实现代理隧道访问一个安全的 web 服务器在防火墙保护下。要运行这个程序,您必须将Java 系统属性设置为适当的值:

```
java -Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=ProxyPortNumber
SSLSocketClientWithTunneling
```

注意:代理规范使用 -D 选项都是可选的。webproxy 替换为您的代理主机的名称,ProxyPortNumber 代表适当的端口号。

程序将从 https://www.verisign.com/index.html 返回的 HTML 源文件。

运行 ClassFileServer

程序 ClassFileServer 有两个文件: ClassFileServer.java 和 ClassServer.java。

运行 ClassFileServer.class,它需要以下参数:

- 端口可以是任何未使用的端口号,例如,您可以使用2001
- docroot 表示服务器上的目录,其中包含您要检索的文件。例如,在Solaris上,您可以使用 /home/userid/(用户标识是指特定的UID),而在Microsoft Windows 系统上,您可以使用 c:\
- TLS 是一个可选参数,表示服务器是使用 SSL 还是 TLS
- true 是一个可选参数,表明客户端身份验证是必须的。这个参数只是在 TLS 参数设置后会咨询

注意: TLS和 true 的参数是可选的。如果你忽略他们,这表明这个当作普通(而不是 TLS)文件服务器使用,没有身份验证,什么也不会发生。这是因为一方(客户端)正试图与TLS协商,而另一个(服务器)没有,所以他们不能交互。

注:服务器预期是 GET 请求的形式 GET /path_to_file

用 ClassFileServer 运行 SSLSocketClientWithClientAuth

您可以使用示例程序 SSLSocketClientWithClientAuth 和 ClassFileServer 设置身份验证通信,客户端和服务器相互验证。您可以运行示例程序在不同的机器上连接到同一个网络,或者你可以在一台机器上用不同的终端窗口或命令提示符窗口运行它们。设置客户端和服务器,执行以下操作:

1.从一台计算机或终端窗口运行程序 ClassFileServer ,如“运行ClassFileServer”。

2.在另一台计算机或终端窗口运行程序 SSLSocketClientWithClientAuth 。SSLSocketClientWithClientAuth 需要以下参数:

- 主机的主机名是您正在使用运行 ClassFileServer 的机器
- 端口是为 ClassFileServer 指定相同的端口
- requestedfilepath 表明文件的路径,你想从服务器检索。你必须给出这个 /filepath 参数。正斜杠被用作 GET 语句的一部分,无关乎使用运行在什么类型的操作系统。形成的声明如下:

```
"GET " + requestedfilepath + " HTTP/1.0"
```

注意:你可以修改其他的 SSLClient* 应用的 GET 命令来连接到本地的运行的 ClassFileServer

说明 HTTPS 连接的示例代码

有两个主要JSSE API 来访问安全通信。一种方法是通过一个 socket 级别API 可用于任意安全通信,说明了 SSLSocketClient,SSLSocketClientWithTunneling,SSLSocketClientWithClientAuth (有或没有 ClassFileServer)示例程序。

第二者更加简单,方法是通过标准 Java URL API。你可以安全地使用 HTTPS URL协议访问具有支持 SSL web服务器通过或计划使用 java.net.URL 类。 支持HTTPS URL方案在许多常见的浏览器实现,它允许访问安全的通信而无需JSSE提供的socket级别API。

一个示例 URL 是 https://www.verisign.com。

信任和密钥管理实现与特定的 HTTPS URL 环境有关。JSSE 实现提供了一个 HTTPS URL 实现。使用不同的 HTTPS 协议实现,设置 java.protocol.handler.pkgs system property 包名称。有关详细信息,请参阅 java.net.URL 类文档。

您可以下载 JSSE 示例文件包括两个示例程序演示如何创建一个 HTTPS 连接。这两个示例程序(URLReader.java 和 URLReaderWithOptions.java) 在 samples/urls 目录。

运行 URLReader

(URLReader.java 程序演示了使用 URL 类访问一个安全的网站。这个程序的输出是 https://www.verisign.com/ 的 HTML 源代码。默认情况下, HTTPS 协议实现包含了 JSSE 的使用。为了使用不同的实现 设置系统属性 java.protocol.handler.pkgs 的值。该值是包含实现的包的名称。

如果您正在运行防火墙后面的示例代码,那么你必须设置 https.proxyHost 和 https.proxyPort 系统属性。例如,要使用代理主机“webproxy”在端口 8080 上,您可以使用 java 命令以下选项:

```
-Dhttps.proxyHost=webproxy
-Dhttps.proxyPort=8080
```

此外,您可以在代码中设置系统属性，使用 java.lang.System 的 setProperty()。例如,你可以在你的程序包括以下行:

```
System.setProperty("java.protocol.handler.pkgs", "com.ABC.myhttpsprotocol");
System.setProperty("https.proxyHost", "webproxy");
System.setProperty("https.proxyPort", "8080");
```

运行 URLReaderWithOptions

URLReaderWithOptions.java 与 URLReader.java 程序类似,除了它可以允许您选择输入任何或所有下列系统属性作为程序运行时的参数:

- java.protocol.handler.pkgs
- https.proxyHost
- https.proxyPort
- https.cipherSuites

运行 URLReaderWithOptions ，这如下命令：

```
java URLReaderWithOptions [-h proxyhost -p proxyport] [-k protocolhandlerpkgs] [-c cip
hersarray]
```

注:多个协议处理程序可以包含在 protocolhandlerpkgs 参数的列表项。多个 SSL 密码套件名称可以包含在 ciphersarray 参数的列表项之间用逗号分隔。可能的密码套件名称使用相同 SSLSocket.getSupportedCipherSuites() 方法返回的。套件的名称取自 SSL 和 TLS 协议规范。

你需要一个 protocolhandlerpkgs 参数，只有当你想使用一个 HTTPS 协议处理器实现 Oracle 提供的默认的一个。

如果您正在运行防火墙后面的示例代码,那么您必须包括代理主机和代理端口。此外,您可以启用包括一个密码套件使列表。

这里有一个例子运行 URLReaderWithOptions 并指定代理主机的“webproxy”在端口8080上:

```
java URLReaderWithOptions -h webproxy -p 8080
```

示例代码演示创建一个安全的 RMI 连接

samples/rmi 目录中的示例代码演示了如何创建一个安全的 Java Remote Method Invocation (RMI) 连接。示例代码是基于 RMI 的例子,基本上是一个“Hello World”示例修改安装和使用一个定制的 RMI socket 工厂。

有关Java RMI的更多信息,请参见Java RMI文档。这个网页指向 Java RMI 相关的 Java RMI 的教程和其他信息。

示例代码演示了 SSLEngine 的使用

SSLEngine 给应用程序开发人员灵活性的选择 I/O 和计算策略。而不是把SSL/TLS 实现自一个特定的 I/O 抽象(如单线程 SSLSocket),SSLEngine 消除了 I/O 和实现自 SSL/TLS的计算约束。

如前所述,SSLEngine 是一种高级的API,不适合日常使用。这里提供一些介绍性的示例代码,帮助说明其使用。第一个演示删除大部分的 I/O 和线程问题,并且关注了许多 SSLEngine 方法。第二个演示是一个更实际的例子展示 SSLEngine 如何结合 Java NIO 创建一个基本的 HTTP/HTTPS 服务器。

运行 SSLEngineSimpleDemo

SSLEngineSimpleDemo 是一个非常简单的应用程序，关注于 SSLEngine 的操作，用来简化了 I/O 和线程问题。这个应用程序创建两个 SSLEngine 对象 通过常见的ByteBuffer对象来进行 SSL/TLS 消息交换的对象。 单一循环连续执行的所有引擎的操作，并且展示了如何建立一个安全连接(握手),应用程序数据传输,以及如何关闭引擎。

SSLEngineResult 提供了大量的关于 SSLEngine 的当前状态的信息。这个例子不解释所有状态的检查。它简化了 I/O 和线程问题,虽然这不是一个很好的用于生产环境的例子，但它对于演示 SSLEngine 的整体功能非常有用的。

运行基于 NIO 的服务器

为了充分利用 SSLEngine 提供的灵活性,您必须首先了解互补的 API,如I/O 以及线程模型。

一个 I/O 模型,在大规模应用程序中开发人员选择使用 NIO SocketChannel 。NIO 引入就是为了解决 java.net.Socket API 固有的可伸缩性问题。SocketChannel 有许多不同的操作模式,包括:

- 阻塞
- 非阻塞
- 非阻塞选择器

示例代码提供了一个基本的 HTTP 服务器,不仅展示了许多新 NIO API,但也展示出 SSLEngine 可以用来创建一个安全的 HTTPS 服务器。服务器不具备生产的质量,但它确实显示许多这样的新 API。

在目录下是一个 README.txt 文件,介绍了服务器,解释了如何构建和配置服务器,并提供代码布局的简要概述。 SSLEngine 用户最感兴趣的文件是ChannelIO.java 和 ChannelIOSecure.java。

注意:在这一节中讨论的服务器的例子包括在 JDK 中。你可以在 jdk-home/samples/nio/server 目录中找到。

使用 JSSE 创建 Keystore

使用 SNI 扩展